Concours Centrale filières MP, PC, PSI et TSI : corrigé

Jean-Loup Carré

Informatique commune – 2015

I.A.1) `[1, 2, 3, 4, 5, 6]`

I.A.2) `[1, 2, 3, 1, 2, 3]`

Attention ! Piège !

Il ne faut pas confondre les listes et les tableaux `numpy`. Sur ces tableaux, les opérations se font terme à terme.

I.B)
```
def smul(n, L):
    return [ x*n for x in L ]
```

I.C.1)
```
def vsom(L1, L2):
    return [ L1[k]+L2[k] for k in range(len(L1)) ]
```

Remarque

On ne vérifie pas que les préconditions (dans le cas présent, que `L1` et `L2` ont la même longueur) des fonctions sont vérifiées.
C'est à celui qui utilise la fonction de prendre ses précautions. Comme le répète Guido van Rossum [1], « *We are all consenting adults here* ».

I.C.2) Voici deux manières différentes de programmer la fonction demandée.

```
def vdiff(L1, L2):
    return [ L1[k]-L2[k] for k in range(len(L1)) ]
```

```
def vdiff(L1, L2):
    return vsom(L1, smul(-1, L2))
```

II.A.1)

$$(S)\begin{cases} z'(t) &= f(y(t)) \\ y'(t) &= z(t) \end{cases}$$

II.A.2) On applique le théorème fondamental de l'analyse, qui dit que si h est dérivable, alors $h(b) - h(a) = \int_a^b h'(t)\mathrm{d}t$ et donc $h(b) = h(a) + \int_a^b h'(t)\mathrm{d}t$.

On l'applique pour $h = y$ et $a = t_i$ et $b = t_{i+1}$ et $h' = z$, on a alors $y(t_{i+1}) = y(t_i) + \displaystyle\int_{t_i}^{t_{i+1}} z(t)\mathrm{d}t$.

En l'appliquant pour $h = z$ et $h' = f(y(t))$ on a l'autre égalité.

II.B.1) L'approximation du sujet donne $\displaystyle\int_{t_i}^{t_{i+1}} z(t)\mathrm{d}t \approx \int_{t_i}^{t_{i+1}} z(t_i)\mathrm{d}t = (t_{i+1} - t_i) \times z(t_i) = h \times z(t_i)$.

On a donc $y(t_{i+1}) \approx y(t_i) + h \times z(t_i)$.
De même $z(t_{i+1}) \approx z(t_i) + h \times f(y(t_i))$.
On en déduit les relations de récurrences :

$$\begin{cases} y_{i+1} &= y_i + h \times z_i \\ z_{i+1} &= z_i + h \times f(y_i) \end{cases}$$

1. Le créateur de Python.

II.B.2)
```
def euler(y0, z0, f, tmax, tmin, n):
    Y=[y0]
    Z=[z0]
    h = (tmax-tmin)/(n-1)
    for i in range(n-1):
        Y.append(Y[i] + h*Z[i])
        Z.append(Z[i] + h*f(Y[i]))
    return Y, Z
```

On a besoin, pour le calcul, des conditions initiales y_0 et z_0, de la fonction f et de h et n. On préfère prendre en argument les paramètres du problème ($t_{\min}$ et $t_{\max}$) et en déduire h plutôt que de prendre h en argument.

II.B.3.a) On a $f(x) = -\omega^2 x$ et donc $g(x) = \frac{\omega^2}{2}x^2$.

On pose alors $E(t) = \frac{1}{2}y'(t)^2 + g(y(t)) = \frac{1}{2}y'(t)^2 + \frac{\omega^2}{2}y(t)^2$ et on dérive.

$E'(t) = y'(t)y''(t) + y'(t)\omega^2 y(t)$ ce qui vaut zéro d'après l'équation (II.1) donc $E(t)$ est une constante qu'on peut noter E.

II.B.3.b) On a : $2E_{i+1} = z_{i+1}^2 + \omega^2 y_{i+1}^2$.

Développons : $z_{i+1}^2 = (z_i - h\omega^2 y_i)^2 = z_i^2 + h^2\omega^4 y_i^2 - 2z_i h\omega^2 y_i$,

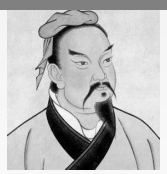
et $\omega^2 y_{i+1}^2 = \omega^2(y_i + hz_i)^2 = \omega^2 y_i^2 + \omega^2 h^2 z_i^2 + 2\omega^2 h z_i y_i$,

d'où $2E_{i+1} = z_i^2 + h^2\omega^4 y_i^2 + \omega^2 y_i^2 + \omega^2 h^2 z_i^2$.

Or $2E_i = z_i^2 + \omega^2 y_i^2$,

donc $2(E_{i+1} - E_i) = h^2\omega^4 y_i^2 + \omega^2 h^2 z_i^2 = h^2\omega^2(\omega^2 y_i^2 + z_i^2) = 2h^2\omega^2 E_i$,
d'où $E_{i+1} - E_i = h^2\omega^2 E_i$.

Conseil stratégique



Sun Tsu

La question est calculatoire et n'est pas utile pour la suite. Si vous vous sentez à l'aise en calcul, faites-là, sinon, n'hésitez pas à la passer.

« Connaissez-vous vous même » *L'art de la guerre de Sun Tsu*, chapitre III.

II.B.3.c) Si l'énergie est conservée, alors elle est la même à chaque instant t_i et $E_{i+1} - E_i = 0$.

Attention ! Piège !

La question est triviale, il n'y a pas de piège. Le seul « piège », c'est qu'on peut croire qu'il y a un piège [2].

II.B.3.d) L'équation « $\frac{1}{2}z^2 + \frac{\omega^2}{2}y^2 = $ constante » est l'équation d'une ellipse.
Lorsque l'énergie est conservée, le graphe est censé être inclus dans une ellipse.

Remarque

Comme les deux axes n'ont pas la même dimension (y et la dérivée de y selon le temps ne sont pas homogènes) la notion de « cercle » n'est pas pertinente.

2. Faire croire à un piège alors qu'il n'y en n'a aucun est le 32e stratagème du classique chinois *Les 36 stratagèmes*.

II.B.3.e) Le graphe n'est pas une ellipse donc l'énergie n'est pas conservée. La courbe s'éloigne du centre de l'ellipse théorique au fur et à mesure que E augmente. C'est pourqui on observe une spirale qui s'élargit.

Attention ! Piège !

Rien n'indique sur le graphique dans quel sens la spirale est parcourue (vers l'intérieur ou vers l'extérieur). On déduit du fait que $E_{i+1} - E_i > 0$ que la spirale est parcourue vers l'extérieur.

II.C.1)
```
def verlet(y0, z0, f, tmax, tmin, n):
    Y=[y0]
    Z=[z0]
    h = (tmax-tmin)/(n-1)
    for i in range(n-1):
        Y.append(Y[i] + h*Z[i] + h**2/2* f(Y[i]))
        Z.append(Z[i] + h*( f(Y[i]) + f(Y[i+1])  ))
    return Y, Z
```

Remarque

Pour obtenir cette fonction, on se contente de modifier légèrement le code de la fonction `euler`.

II.C.2.a) Posons $\overline{y_{i+1}} = y_i + hz_i$ et $\overline{z_{i+1}} = z_i + hf_i$ pour se ramener au cas d'Euler.

$$z_{i+1} = z_i + \frac{h}{2}(f_i + f_{i+1}) = \underbrace{z_i + hf_i}_{\overline{z_{i+1}}} - \frac{\omega^2}{2}h^2 z_i + \mathscr{O}(h^3)$$

$$y_{i+1} = \overline{y_{i+1}} + \frac{h^2}{2} f_i$$

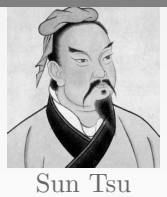
$$\begin{aligned} 2(E_{i+1} - E_i) &= z_{i+1}^2 + y_{i+1}^2 - z_i^2 - \omega^2 y_i^2 \\ &= \overline{z_{i+1}}^2 - \omega^2 h^2 z_i \overline{z_{i+1}} + \omega^2 \overline{y_{i+1}}^2 + \omega^2 \overline{y_{i+1}} h^2 f_i - z_i^2 - \omega^2 y_i^2 + \mathscr{O}(h^3) \end{aligned}$$

D'après la question II.B.3.b), $\overline{z_{i+1}}^2 + \omega^2 \overline{y_{i+1}}^2 = 2h^2\omega^2 E_i$. De plus, $\overline{y_{i+1}} = y_i + \mathscr{O}(h)$ et $\overline{z_{i+1}} = z_i + \mathscr{O}(h)$. On a donc

$$\begin{aligned} 2(E_{i+1} - E_i) &= 2\omega^2 h^2 E_i - \omega^2 h^2 z_i z_i + \omega^2 y_i h^2 f_i + \mathscr{O}(h^3) \\ &= \omega^2 h^2 (z_i^2 + \omega^2 y_i^2) - \omega^2 h^2 z_i z_i + \omega^2 y_i h^2 \times (-\omega^2 y_i) + \mathscr{O}(h^3) \\ &= \mathscr{O}(h^3) \end{aligned}$$

En conlusion $E_{i+1} - E_i = \mathscr{O}(h^3)$

Conseil stratégique

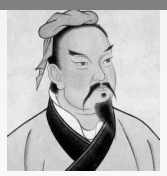


Sun Tsu

Le schéma de Verlet étant similaire au schéma d'Euler, on a ici introduit $\overline{y_{i+1}}$ et $\overline{z_{i+1}}$ pour pouvoir utiliser les résultats de la question II.B.3.b) : on évite ainsi de faire deux fois les mêmes calculs.

Il est important d'utiliser les grand O de sorte à simplifier les calculs.

Conseil stratégique


Sun Tsu

La question est chronophage et rapporte peu. Gardez là pour la fin.
« *Une armée peut être comparée à l'eau ; l'eau épargne les lieux élevés et gagne les creux ; une armée contourne la force et attaque l'inconsistance* ». L'art de la guerre (VI.27) selon Sun Tsu.

II.C.2.b) La courbe est fermée, on observe une ellipse. L'énergie est beaucoup mieux conservée par le schéma de Verlet.

II.C.2.c) Le schéma de Verlet est plus approprié qu'Euler pour l'étude de systèmes conservatifs.

Remarque

Dans un système non-conservatif, l'erreur sur l'énergie peut être négligeable devant la variation d'énergie. Ce n'est pas possible dans un système conservatif, car la variation d'énergie est nulle.

III.A.1) $\overrightarrow{F_j} = \sum_{\substack{k=0 \\ k \neq j}}^{N-1} \overrightarrow{F_{k/j}}$

III.A.2) On introduit une fonction auxiliaire calculant la norme d'un vecteur.

```
def norm(v):
    return (v[0]**2+v[1]**2+v[2]**2)**0.5
```

On introduit une constante globale pour G.

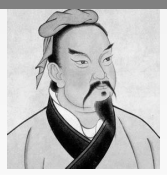
```
G = 6.67*10**-11

def force2(m1, p1, m2, p2):
    p1p2 = vdiff(p2, p1)
    r = norm(p1p2)
    return smul(G*m1*m2/r**3, p1p2)
```

Remarque

On pourrait aussi définir `G` comme suit : `G = 6.67e-11`

Conseil stratégique


Sun Tsu

On utilise les fonctions définies au début du sujet. On ne reprogramme pas ce qui a déjà été programmé.

III.A.3)
```
def forceN(j, m, pos):
    N = len(pos)
    F = [0, 0, 0]
    for k in range(N):
```

```
            if k != j:
                F = vsom(F, force2(m[j], pos[j], m[k], pos[k]))
        return F
```

III.B.1) `position[i]` et `vitesses[i]` sont deux listes de listes de 3 éléments. Elle représentent la liste, à l'instant i, de, respectivement, les positions de chaque corps et les vitesses de chaque corps.

III.B.2)
```
def pos_suiv(m, pos, vit, h):
    N = len(pos)
    pos2 = [None] * N
    for j in range(N):
        euler = smul(h, vit[j]) # terme en h de la méthode d'Euler
        acc = smul(1/m[j],forceN(j, m, pos)) # accélération du corps j
        verlet = smul(h**2/2, acc) # terme en h**2
        pos2[j] = vsom(pos[j], vsom(euler, verlet))
    return pos2
```

III.B.3)
```
def etat_suiv(m, pos, vit, h):
    N = len(pos)
    vit2 = [None] * N
    pos2 = pos_suiv(m, pos, vit, h) # nouvelles positions
    for j in range(N):
        acc = smul(1/m[j],forceN(j, m, pos))
        acc2 = smul(1/m[j],forceN(j, m, pos2)) # nouvelle accélération
        delta_z = smul(h/2, vsom(acc, acc2))
        vit2[j] = vsom(vit[j], delta_z)
    return pos2, vit2
```

III.B.4.a) Le graphique fait apparaître, aux erreurs de mesure près, une relation affine entre $\ln(N)$ et $\ln(\tau_N)$.
En prenant quelques points sur le graphique, on trouve $\ln(\tau_N) \approx 2\ln(N) - 9.5$

III.B.4.b) On a alors $\tau_N \approx e^{2\ln(N)} \times e^{-9.5} = N^2 \times e^{-9.5} = \mathcal{O}(N^2)$.
On conjecture à partir du graphique une complexité en $\mathcal{O}(N^2)$.

III.B.5.a) La fonction `forceN` est en $\mathcal{O}(N)$. Les appels aux fonctions `vsom` et `smul` sont en temps constant, car elles sont appelées sur des listes de longueur 3. On passe N fois dans le `for` de `pos_suiv`, donc la complexité de `pos_suiv` est en $\mathcal{O}(N^2)$.
De même, on passe N fois dans le `for` de `etat_suiv` qui appelle `forceN`. Donc ce second `for` coûte $\mathcal{O}(N^2)$ et `etat_suiv` a une complexité totale de $\mathcal{O}(N^2) + \mathcal{O}(N^2) = \mathcal{O}(N^2)$.

III.B.5.b) On a validé la conjecture.

IV.A)
```
SELECT masse FROM corps
```

IV.B.1)
```
SELECT COUNT(DISTINCT id_corps)
  FROM etat
  WHERE datem <= tmin()
```

IV.B.2)
```
SELECT id_corps, MAX(datem) as dater_der
  FROM etat
  WHERE datem <= tmin()
  GROUP BY id_corps
```

IV.B.3)
```
SELECT masse, x, y, z, vx, vy, vz
  FROM etat JOIN date_mesure ON etat.id_corps = date_mesure.idcorps
  JOIN corps ON etat.id_corps = corps.id_corps
  WHERE datem = dateder AND masse >= masse_min()
    AND -arete()/2 < x AND x < arrete()/2
    AND -arete()/2 < y AND y < arrete()/2
    AND -arete()/2 < z AND z < arrete()/2
  ORDER BY x*x+y*y+z*z  -- La distance au carré est plus rapide
                       -- à calculer que la distance.
```

Remarque

Il est possible de simplifier la requête en utilisant la fonction `ABS` et en utilisant le mot-clef `USING`.

```
SELECT masse, x, y, z, vx, vy, vz
  FROM etat JOIN date_mesure USING(id_corps)
  JOIN corps USING(id_corps)
  WHERE datem = dateder AND masse >= masse_min()
    AND ABS(x) < arrete()/2
    AND ABS(y) < arrete()/2
    AND ABS(z) < arrete()/2
  ORDER BY x*x+y*y+z*z
```

IV.C) On introduit une fonction auxiliaire `convert` qui multiplie tous les vecteurs d'une liste de vecteurs `L` par une constante `K`. Cette fonction nous servira par la suite à convertir les UA en mètres et les km/s en m/s.

```
def convert(L, K):
    return [ smul(K, v) for v in L ]
```

On peut alors écrire la fonction de simulation :

```
def simulation_verlet(delta, n):
    pos = convert(p0, 1.5e11) #p0 est une globale
    vit = convert(v0, 10**3)
    position = [pos]
    for k in range(n):
        pos, vit = etat_suiv(m, pos, vit, delta)
        position.append(pos)
    return position
```